More Instruction Level Parallelism Explains the Actual Efficiency of Compensated Algorithms

Philippe Langlois, Nicolas Louvet

To cite this version:
Philippe Langlois, Nicolas Louvet. More Instruction Level Parallelism Explains the Actual Efficiency of Compensated Algorithms. 11 pages. 2007. <hal-00165020>

HAL Id: hal-00165020
https://hal.archives-ouvertes.fr/hal-00165020
Submitted on 24 Jul 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
More Instruction Level Parallelism Explains the Actual Efficiency of Compensated Algorithms

Philippe Langlois and Nicolas Louvet
langlois@univ-perp.fr, nicolas.louvet@univ-perp.fr

Abstract—The compensated Horner algorithm and the Horner algorithm with double-double arithmetic improve the accuracy of polynomial evaluation in IEEE-754 floating point arithmetic. Both yield a polynomial evaluation as accurate as if it was computed with the classic Horner algorithm in twice the working precision. Both algorithms also share the same low-level computation of the floating point rounding errors and cost a similar number of floating point operations. We report numerical experiments to exhibit that the compensated algorithm runs at least twice as fast as the double-double one on modern processors. We propose to explain such efficiency by identifying more instruction level parallelism in the compensated implementation. Such property also applies to other compensated algorithms for summation, dot product and triangular linear system solving. More generally this paper illustrates how this kind of performance analysis may be useful to highlight the actual efficiency of numerical algorithms.

Index Terms—Accurate polynomial evaluation, Horner algorithm, compensated Horner algorithm, floating point arithmetic, IEEE-754 standard, instruction level parallelism, performance evaluation.

I. INTRODUCTION

In this paper, we consider polynomial evaluation in floating point arithmetic restricted to entries and polynomial coefficients being floating point values. Such cases appear for example when evaluating elementary functions [1] and in geometric computations where accurate polynomial evaluation is crucial [2], [3].

A. Accurate Polynomial Evaluation

The following inequality bounds the accuracy of the floating point result \( \hat{p}(x) \) of the polynomial evaluation \( p(x) \), for example with the classic Horner algorithm. We have

\[
\frac{|p(x) - \hat{p}(x)|}{|p(x)|} \leq \text{cond}(p, x) \times O(u),
\]

where \( u \) is the computing precision and the condition number \( \text{cond}(p, x) \) is a scalar larger than 1 that only depends on the entry \( x \) and on \( p \) coefficients —its expression will be given further. Hence the computed value \( \hat{p}(x) \) suffers from less exact digits than what the computing precision provides. This loss of accuracy may be arbitrarily large as evaluating the polynomial \( p \) at the \( x \) entry is more ill-conditioned, as for example in the neighborhood of a multiple root.

When the computing precision \( u \) is not sufficient (compared to \( \text{cond}(p, x) \)) to guarantee a desired accuracy in \( \hat{p}(x) \), several solutions implementing a computation with more bits exist. Double-double algorithms are well-known and well-used solutions to simulate twice the IEEE-754 double precision [4], [5]. The compensated Horner algorithm introduced in [6] is an alternative to the Horner algorithm implemented with double-double arithmetic. In both cases the accuracy of computed \( \hat{p}(x) \) is improved and now verifies

\[
\frac{|p(x) - \hat{p}(x)|}{|p(x)|} \leq u + \text{cond}(p, x) \times O(u^2).
\]

Comparing to Relation (1), this relation means that the computed value is now as accurate as the result of the Horner algorithm performed in twice the working precision with a final rounding back to this working precision —the same behavior is mentioned in [7] for compensated summation and dot product.

As for Relation (1) the accuracy of the compensated result still depends on the condition number and may be arbitrarily bad for ill-conditioned polynomial evaluations. Nevertheless, this bound tells us that the compensated Horner algorithm may yield a full precision accuracy for not too ill-conditioned polynomials, that is for \( p \) and \( x \) such that the second term \( \text{cond}(p, x) \times O(u^2) \) is small compared to the working precision \( u \). In [8] we prove that the compensated evaluation is faithfully rounded for condition numbers up to \( O(u^{-1}) \). By faithful rounding we mean that the computed result is one of the two floating point neighbors of the exact result \( p(x) \). We also provide a dynamical test to answer to the question “is the computed compensated result a faithful rounding of the exact evaluation?” thanks to a computable and validated bound for the final absolute error in \( \hat{p}(x) \).

B. Previous Results and Motivation for Efficiency Analysis

Compensated Horner evaluation is fast. By fast we mean that it runs at least twice as fast as the double-double Horner counterpart still providing the same output accuracy. The implementation core of these “double-like” algorithms is the computation of the rounding errors generated by the floating point operators. For compensated implementations, these rounding errors are used to correct the result of the original algorithm. Such low-level computation depends on the arithmetic attributes. In [9] we present experimental results to exhibit how to benefit from the fused multiply and add operator. It appears that FMA should be avoided in the evaluation part of the compensated algorithm but preferred when computing the rounding errors. Measures showing the efficiency of the compensated Horner algorithm also when FMA is not available are briefly presented in [8].

In both cases these experiments illustrate the practical efficiency of the compensated algorithm we announced before. Nevertheless we were not able to explain why the measured overhead factor introduced by the compensated evaluation...
algorithm is significantly better than the one introduced by its double-double counterpart. Counting of floating point operations is still commonly used in the field of numerical analysis to compare the performances of different numerical algorithms. But this classic technique is clearly not sufficient to answer to the open question we address here, as this is summarized in Table I. How to explain that computed Horner actually runs twice as fast as the double-double Horner whereas their flop counts are very similar? Let us remark that the same property is identified but still unexplained for summation and dot product in [7].

In this paper we propose to answer to this open question presenting how the actual efficiency of the compensated Horner algorithm can be explained thanks to its instruction level parallelism (ILP). Quoting Hennessy and Patterson [10, p.172], “all processors since about 1985 … use pipelining to overlap the execution of instructions and improve performances. This potential overlap among instructions is called instruction-level parallelism since the instructions can be evaluated in parallel.”

We propose a detailed analysis of the ILP of compensated Horner and Horner with double-double algorithms. We quantify the average number of instructions that can be theoretically executed in one clock cycle on an ideal processor. This ideal processor is one where all the artificial constraints on ILP are removed [10, p.240]. In this context, the theoretical IPC (instructions per clock) is about six times better for the compensated Horner algorithm than for its double-double counterpart. Every double-double arithmetic operation ends with a renormalization step [4], [5]. We also show that avoiding these renormalization steps the compensated Horner algorithm presents more ILP than the Horner algorithm with double-double arithmetic. We conclude that the compensated algorithm exhibits more potential to benefit from the superscalar facilities of modern processors. In our point of view, this gives a qualitative explanation of its practical efficiency.

C. Outline

The paper is organized as follows. In Section II we describe the main steps from the classic Horner algorithm to the compensated Horner algorithm. This Section summarizes some results already presented in [8]: error free transformations of arithmetic operations, extension to an error free transformation of the Horner polynomial evaluation, final correction and the corresponding theoretical accuracy bound. We present experimental measures of the running times for the compensated Horner algorithm and the challenging Horner with double-double arithmetic in Section III. Since the classic flop count fails to explain these observed results we devote the last Section IV to a detailed comparison of these two algorithms. We introduce the notions of ILP and IPC on an ideal processor. Then we highlight the common parts within the compensated and the double-double Horner algorithms and prove that the compensated implementation benefits for more ILP than the double-double one. Appendix contains all the measures previously analyzed in Section III.

D. Notations and Hypothesis

Throughout the paper, we assume a floating point arithmetic compliant with the IEEE-754 floating point standard [11]. We constraint all the computations to be performed in one working precision, with the “round to the nearest” rounding mode. We also assume that no overflow nor underflow occurs during the computations. Next notations are standard (see [12, chap. 2] for example). F is the set of all normalized floating point numbers and u denotes the unit roundoff, that is half the spacing between 1 and the next representable floating point value. For IEEE-754 double precision with rounding to the nearest, we have $u = 2^{-53} \approx 1.11 \cdot 10^{-16}$.

The symbols $\oplus$, $\odot$, $\otimes$ and $\ominus$ represent respectively the floating point addition, subtraction, multiplication and division. For more complicated arithmetic expressions, fl$(\cdot)$ denotes the result of a floating point computation where every operation inside the parenthesis is performed in the working precision. So we have for example, $a \oplus b = \text{fl}(a + b)$.

When no underflow nor overflow occurs, the following standard model describes the accuracy of every considered floating point computation. For two floating point numbers $a$ and $b$ and for $\circ$ in $\{+, -, \times, /\}$, the floating point evaluation fl$(a \circ b)$ of $a \circ b$ is such that

$$\text{fl}(a \circ b) = (a \circ b)(1 + \varepsilon_1) = (a \circ b)/(1 + \varepsilon_2), \text{ with } |\varepsilon_1|, |\varepsilon_2| \leq u.$$  

(3)

It is classic to keep track of the $(1 + \varepsilon)$ factors when nesting $k$ arithmetic operations using $\gamma_k := ku/(1 - ku)$. We have $\prod_{i=1}^{k}(1 + \varepsilon_i) \leq 1 + \gamma_k$ [12, chap. 3]. When using these notations, we always implicitly assume $ku < 1$ and $k > 0$.

II. FROM HORBNE TO COMPENSATED HORBNE ALGORITHM

The compensated Horner algorithm improves the classic Horner iteration by computing a correct term to compensate the rounding errors the Horner iteration generates in floating point arithmetic. Main results about compensated Horner algorithm are summarized in this section and may be skipped if [8] is already known.

A. Polynomial evaluation and Horner algorithm

The classic condition number of the evaluation of $p(x) = \sum_{i=0}^{n} a_i x^i$ at a given entry $x$ is [13]

$$\text{cond}(p, x) = \frac{\sum_{i=0}^{n} \left| a_i \right| \left| x \right|^i}{\left| \sum_{i=0}^{n} a_i x^i \right|} := \tilde{p}(x).$$  

(4)
For any floating point value \( x \) we denote by \( \text{Horner}(p,x) \) the result of the floating point evaluation of the polynomial \( p \) at \( x \) using the classic Horner algorithm recalled below.

**Algorithm 1 Horner algorithm**

```plaintext
function \( r_0 = \text{Horner}(p,x) \)

\[
\begin{align*}
    r_n &= a_n \\
    \text{for } i = n - 1 : -1 : 0 \\
    r_i &= r_{i+1} \otimes x \oplus a_i \\
\end{align*}
\]

end
```

The accuracy of Algorithm 1 verifies introductory inequality (1) with \( O(u) = \gamma_{2n} \) and previous condition number (4). Clearly, the condition number \( \text{cond}(p,x) \) can be arbitrarily large. In particular, when \( \text{cond}(p,x) > \gamma_{2n}^{-1} \), we cannot guarantee that the computed result \( \text{Horner}(p,x) \) contains any correct digit.

More accuracy can be reached at the same computing precision thanks to error free transformation (EFT). We review well known results concerning the EFT of the elementary floating point operations thanks to error free transformation (EFT). We review well known that a dynamic sorting ruins the actual performances.

**Algorithm 2 EFT of the sum of two floating point numbers**

```plaintext
function \( [x, y] = \text{TwoSum}(a, b) \)

\[
\begin{align*}
    x &= a \oplus b \\
    z &= x \odot a \\
    y &= (a \odot (x \odot z)) \oplus (b \odot z) \\
\end{align*}
\]

end
```

We notice that algorithms \( \text{TwoSum} \) and \( \text{TwoProd} \) only require well optimizing floating point operations. They apply for the IEEE rounding to the nearest rounding mode. They do not use branches, nor access to the mantissa that can be time-consuming. \( \text{FastTwoSum} \) costs less flop than \( \text{TwoSum} \) but only applies to sorted entries. When this condition is not \textit{a priori} satisfied, it is well known that a dynamic sorting ruins the actual performances.

**Algorithm 3 EFT of the product of two floating point numbers**

```plaintext
function \( [x, y] = \text{TwoProd}(a, b) \)

\[
\begin{align*}
    x &= a \odot b \\
    \% \text{ splitting of } a \text{ and } b \text{ to high and low parts} \\
    \% \text{ splitter } = (1 + 2^{l}) \text{ is a predefined constant,} \\
    \% \text{ with } t \text{ the mantissa length} \\
    a_s &= \text{splitter} \odot a; \quad b_s = \text{splitter} \odot b \\
    a_k &= a_s \odot (a_s \odot a); \quad b_k = b_s \odot (b_s \odot b) \\
    a_l &= a - a_h; \quad b_l = b - b_h \\
    \% \text{ rounding error in } a \odot b \\
    y &= a_l \odot b_l \odot ((a_s \odot b_s) \odot a_l \odot b_l) \odot (a_h \odot b_h) \\
\end{align*}
\]

end
```

of \( \text{FastTwoSum} \) on superscalar processors [7]. We also mention that a significant improvement of \( \text{TwoProd} \) is defined when a Fused-Multiply-and-Add operator is available as Intel Itanium or IBM PowerPC [16]. We detailed how to benefit from such instruction in [9].

**Algorithm 4 EFT of the sum of two sorted floating point numbers**

```plaintext
function \( [x, y] = \text{FastTwoSum}(a, b) \)

\[
\begin{align*}
    \% \text{ Assume } |a| \geq |b| \\
    x &= a \odot b \\
    y &= (a \odot x) \odot b \\
\end{align*}
\]

end
```

**Algorithm 5 EFT for the Horner algorithm**

```plaintext
function \( [r_0, p_\sigma, p_\sigma] = \text{EFTHorner}(p,x) \)

\[
\begin{align*}
    r_n &= a_n \\
    \text{for } i = n - 1 : -1 : 0 \\
    [p_{i}, \sigma_i] &= \text{TwoProd}(r_{i+1}, x) \\
    [r_{i}, \sigma_i] &= \text{TwoSum}(p_i, a_i) \\
    \text{Let } \sigma_i \text{ be the coefficient of degree } i \text{ in } p_\sigma \\
    \text{Let } \sigma_i \text{ be the coefficient of degree } i \text{ in } p_\sigma \\
\end{align*}
\]

end
```

Theorem 1 ([8]): Let \( p(x) = \sum_{i=0}^{n} a_i x^i \) be a polynomial of degree \( n \) with floating point coefficients, and let \( x \) be a floating point value. Then Algorithm 5 computes both \( \text{Horner}(p,x) \) and two polynomials \( p_\sigma \) and \( p_\sigma \) of degree \( n - 1 \) with floating point coefficients, such that \([\text{Horner}(p,x), p_\sigma, p_\sigma] = \text{EFTHorner}(p,x) \). If no underflow occurs, the polynomial evaluation verifies

\[
p(x) = \text{Horner}(p,x) + (p_\sigma + p_\sigma)(x).
\]

Relation (5) means that algorithm \( \text{EFTHorner} \) is an EFT for polynomial evaluation with the Horner algorithm.

**D. Compensated Horner algorithm**

From Theorem 1 the forward error in the floating point evaluation of \( p(x) \) with the Horner algorithm is

\[
e = p(x) - \text{Horner}(p,x) = (p_\sigma + p_\sigma)(x),
\]

where \( e \) is the forward error of the floating point evaluation of \( p(x) \) with the Horner algorithm.
where both polynomials $p_\pi$ and $p_\sigma$ are exactly identified by EFTHorner (Algorithm 5) — this latter also computes Horner($p, x$). Therefore, the key of the compensated algorithm is to compute, in the working precision, the approximate Horner($p_\pi \oplus p_\sigma, x$) of the final error $c$ and then a corrected result

$$\tau = \text{Horner}(p, x) \oplus \text{Horner}(p_\pi \oplus p_\sigma, x).$$

We say that Horner($p_\pi \oplus p_\sigma, x$) is a correcting term for Horner($p, x$). The compensated result $\tau$ is expected to be more accurate than Horner($p, x$) as proved in next section. The next Algorithm 6 implements the compensated Horner algorithm within only one loop inlining the computation of the Horner EFT (Algorithm 5), the computation of the correcting term Horner($p_\pi \oplus p_\sigma, x$) and the final correction.

**Algorithm 6** Compensated Horner algorithm

```c
function \( \tau = \text{CompHorner}(P, x) \)
    \( r_n = a_i; c_n = 0 \)
    for \( i = n - 1 : -1 : 0 \)
      \[
      \begin{align*}
      \{a_i, \pi_i\} &= \text{TwoProd}(r_{i+1}, x) \\
      \{r_i, \sigma_i\} &= \text{TwoSum}(p_i, a_i) \\
      c_i &= c_{i+1} \otimes x \oplus (\pi_i \oplus \sigma_i)
      \end{align*}
      \]
    end
    \%
    Here \( r_0 = \text{Horner}(p, x) \), 
    \%
    and \( c_0 = \text{Horner}(p_\pi \oplus p_\sigma, x) \) 
    \( \tau = r_0 \oplus c_0 \)
```

**E. Accuracy of the Compensated Horner Algorithm**

Next result proves that the result of a polynomial evaluation computed with the compensated Horner algorithm (Algorithm 6) is as accurate as if computed by the Horner algorithm using twice the working precision and then rounded to the working precision.

**Theorem 2** ([8]): Consider a polynomial $p$ of degree $n$ with floating point coefficients, and $x$ a floating point value. If no underflow occurs,

$$|\text{CompHorner}(p, x) - p(x)| \leq u|p(x)| + \gamma_{2n}^2 \tilde{p}(x).$$

It is interesting to interpret the previous theorem in terms of the condition number of the polynomial evaluation of $p$ at $x$. Combining the error bound (6) with the condition number (4) of polynomial evaluation gives the precise writing of our introductory inequality (2).

$$|\text{CompHorner}(p, x) - p(x)| \leq u + \gamma_{2n}^2 \text{cond}(p, x).$$

Since $\gamma_{2n}^2 = O(u^2)$ relation (7) essentially tells us that the compensated result is as accurate as if computed by the classic Horner algorithm in twice the working precision, with a final rounding back to the working precision [8].

Fig. 1 illustrates the accuracy behavior of Horner and CompHorner w.r.t. the condition number. More detailed experiments may be found in [8]. We generate polynomials of degree 50 whose condition numbers vary from about $10^2$ to $10^{35}$. We see that even for small condition numbers we already lose some accuracy with the Horner evaluation. We also observe that the compensated algorithm exhibits the expected behavior: as long as the condition number is smaller than $u^{-1}$, the relative error is of the order of the working precision $u$. Then, for condition numbers between $u^{-1}$ and $u^{-2}$, this relative error degrades to no accuracy at all. As usual, the *a priori* error bound (7) appears to be pessimistic by many orders of magnitude.

**III. Experimental Results for Performance Analysis**

Now we start to focus the open question that motivates this paper presenting our implementations and the corresponding measured running-times for Compensated Horner algorithm and Horner algorithm with double-double.

**A. Implementation of the Compensated Horner Algorithm**

Since every loop of Horner (Algorithm 1) includes a multiplication by $x$, every loop of CompHorner (Algorithm 6) introduces one TwoProd applied to the same $x$. Hence the split of $x$ is only performed once (out of the loop) to reduce the flop count. The following C code implements this simplification. Algorithm CompHorner now requires $22n + 5$ flops.

```c
double CompHorner(double *p, unsigned int n, double x) {
    double p, r, a, pi, sig; x_hi, x_lo, hi, lo, t;
    int i;
    /\* (x_hi, x_lo) = Split(x) */
    t = x*xsplitter; x_hi = t-(t-x); x_lo = x-x_hi;
    r = p[n]; c = 0.0;
    for(i=n-1; i>=0; i--) {
        /\* (p, pi) = TwoProd(r, x) */
        p = r*x; sh
        t = r*splitter; hi = t-(t-r); lo = r-hi;
        pi = ((hi*x_hi-p)+hi*x_lo)+lo*x_hi)+lo*x_lo;
        /\* (s, sigma) = TwoSum(p, P[i]) */
        r = p+P[i];
        t = r-p;
        sig = (p-(r-t)) + (P[i]-t);
        /\* Computation of the correcting term */
        c = c*x*(pi+sig);
    }
    return(r+c);
}
```

**B. Horner Algorithm with Double-Double Computation**

In Algorithm 7, we implement the Horner algorithm performed with double-double arithmetic.

**Algorithm 7** Horner algorithm with double-doubles

```c
function r = DDDHorner(P, x)
sh = a_i; sl = 0
for i = n - 1 : -1 : 0
    \%
    double-double = double-double \times double:
    \%
    \% (ph_i, pl_i) = (sh_{i+1}, sl_{i+1}) \otimes x
    \%
    [th, tl] = TwoProd(sh_{i+1}, x)
    \%
    tl = sl_{i+1} \otimes x \oplus tl
    \%
    [ph_i, pl_i] = FastTwoSum(th, tl)
    \%
    end
    r = sh_0
```

\[
\begin{align*}
\text{\% double-double} &= \text{double-double} \times \text{double}: \\
\text{% (ph_i, pl_i)} &= (sh_{i+1}, sl_{i+1}) \otimes x \\
[th, tl] &= \text{TwoProd}(sh_{i+1}, x) \\
[th, tl] &= \text{TwoSum}(ph_i, a_i) \\
tl &= tl \oplus pl_i \\
[sh_i, sl_i] &= \text{FastTwoSum}(th, tl)
\end{align*}
\]

end

\[
r = sh_0
\]
Double-doubles are managed as proposed by the authors of [5], [17]. For our purpose, it suffices to know that a double-double number $a$ is the pair $(ah, al)$ of IEEE-754 floating point numbers with $a = ah + al$ and $|al| \leq u|ah|$. As we will see in the sequel, this property requires a renormalization step after every arithmetic operation with double-double values. To implement the Horner algorithm using the double-double format, we only need two basic operations: i) the product of a double-double number by a double number, and ii) the addition of a double number to a double-double number. These operations are represented in boxes in Algorithm 7. We notice that every double-double operation performs a final renormalization step using algorithm FastTwoSum (Algorithm 4). This renormalization is compulsory to ensure that the computed result of the corresponding double-double operation is still a valid double-double number. For example, the line $[sh_i, sl_i] = \text{FastTwoSum}(th, tl)$ in Algorithm 4 ensures that the floating point pair $(sh_i, sl_i)$ actually satisfies $|sl_i| \leq u|sh_i|$ and so is a valid double-double number.

We also provide hereafter a C code implementation of algorithm DDHorner. We count that algorithm DDHorner requires $28n + 4$ flops.

```c
double DDHorner(double *P, unsigned int n, double x) {
    double r_h, r_l, t_h, t_l, x_hi, x_lo, hi, lo, t;
    int i;
    /* (x_hi, x_lo) = Split(x) */
    t = x*splitter; x_hi = t-(t-x); x_lo = x-x_hi;
    r_h = P[n]; r_l = 0.0;
    for(i=n-1; i>=0; i--){
        /* (r_h, r_l) = (r_h, r_l) * x */
        t = r_h*splitter; hi = t-(t-r_h); lo = (r_h-hi);
        t_h = r_h*x;
        t_l = (((hi*x_hi-t_h)+hi*x_lo)+lo*x_hi)+lo*x_lo;
        t_l += r_l*x;
        r_h = t_h+t_l;
        r_l = (t_h-r_h)+t_l;
        /* (r_h, r_l) = (r_h, r_l) + P[i] */
        t_h = r_h+P[i];
        t = t_h-r_h;
        t_l = ((r_h-(t-h-t))+(P[i]-t))+r_l;
        r_h = t_h+t_l;
    }
    return(r_h);
}
```

C. Experimental results

All our experiments are performed using IEEE-754 double precision, and the algorithms are implemented in C code. The experimental environments are listed in Table II.

<table>
<thead>
<tr>
<th>env.</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(I)</td>
<td>Intel Pentium 4, 3 GHz, GCC 4.1.2 -std=c99 -march=pentium4 -mfpmath=387 -O3 -funroll-all-loops</td>
</tr>
<tr>
<td>(II)</td>
<td>Intel Pentium 4, 3 GHz, ICC 9.1 -c99 -mtune=pentium4 -O3 -funroll-loops -mp1</td>
</tr>
<tr>
<td>(III)</td>
<td>Intel Pentium 4, 3 GHz, GCC 4.1.2 -std=c99 -march=pentium4 -mfpmath=sse -O3 -funroll-all-loops</td>
</tr>
<tr>
<td>(IV)</td>
<td>Intel Pentium 4, 3 GHz, ICC 9.1 -c99 -mtune=pentium4 -msse2 -O3 -funroll-loops -fp-model source</td>
</tr>
<tr>
<td>(V)</td>
<td>AMD Athlon 64 3200+, 2 GHz, GCC 4.1.2 -std=c99 -march=athlon64 -msse2 -O3 -funroll-all-loops</td>
</tr>
<tr>
<td>(VI)</td>
<td>Itanium 2, 1.5 GHz, GCC 4.1.1 -c99 -mtune-itanium2 -O3 -funroll-all-loops</td>
</tr>
<tr>
<td>(VII)</td>
<td>Itanium 2, 1.5 GHz, ICC 4.1.1 -c99 -mtune-itanium2 -O3 -funroll-all-loops</td>
</tr>
</tbody>
</table>

In this table, GCC denotes the GNU Compiler Collection and ICC the Intel C Compiler. In environments (I) to (V), no FMA instruction is available, so we use the C codes presented in the previous subsections to implement algorithms CompHorner and DDHorner. In environments (VI) and (VII), that is on the Intel Itanium architecture which provides a FMA instruction, we use
the improvements of these algorithms presented in [9]. Anyway we use the same programming techniques for the implementations of the routines CompHorner and DDHorner. All timings are done with the cache warmed to minimize the memory traffic overhead.

Our measures are performed with 39 polynomials whose degrees vary from 10 to 200 by step of 5. The coefficients and the argument of these polynomials are randomly generated. For every algorithm and every degree, we measure the ratio of the computing time over the computing time of the Horner algorithm.

We display the average value of these ratios for CompHorner and DDHorner in Table III. All detailed results for considered environments are presented in the Appendix.

<table>
<thead>
<tr>
<th>env.</th>
<th>CompHorner/Horner</th>
<th>DDHorner/Horner</th>
</tr>
</thead>
<tbody>
<tr>
<td>(I)</td>
<td>2.8</td>
<td>8.5</td>
</tr>
<tr>
<td>(II)</td>
<td>2.7</td>
<td>9.0</td>
</tr>
<tr>
<td>(III)</td>
<td>3.0</td>
<td>8.9</td>
</tr>
<tr>
<td>(IV)</td>
<td>3.2</td>
<td>9.7</td>
</tr>
<tr>
<td>(V)</td>
<td>3.2</td>
<td>8.7</td>
</tr>
<tr>
<td>(VI)</td>
<td>2.9</td>
<td>7.0</td>
</tr>
<tr>
<td>(VII)</td>
<td>1.5</td>
<td>5.9</td>
</tr>
</tbody>
</table>

First we notice that the measured slowdown factors are always significantly smaller than expected if the flop count is only considered. We have from previous flop counts,

\[
\frac{\text{CompHorner}}{\text{Horner}} = \frac{22n + 5}{2n} \approx 11,
\]

whereas the compensated algorithm CompHorner is only about 3 times slower than the classic Horner algorithm. The same remark applies to algorithm DDHorner for which

\[
\frac{\text{DDHorner}}{\text{Horner}} = \frac{28n + 4}{2n} \approx 14,
\]

and that appears to be only about 8 times slower than the Horner algorithm.

These results also show that the compensated algorithm is more than twice faster than the Horner algorithm with double-double computation while, as already noticed, previous comparison suggests

\[
\frac{\text{DDHorner}}{\text{CompHorner}} \approx \frac{28n + 4}{22n + 5} \approx 1.3.
\]

Flop counts are therefore not sufficient to explain such behaviors.

IV. MORE ILP IN CompHorner EXPLAINS ITS ACTUAL PERFORMANCE

It is well known that most modern processors are capable of executing several instructions concurrently. Quoting Hennessy and Patterson [10, p.172]:

All processors since about 1985, including those in the embedded space, use pipelining to overlap the execution of instructions and improve performances. This potential overlap among instructions is called instruction-level parallelism since the instructions can be evaluated in parallel.

As explained in Section I, the term instruction-level parallelism refers to the degree to which the instructions of a program can be executed in parallel. Real programs are usually written in a serial fashion, using high-level languages such as C and Fortran. To exploit the implicit parallelism available among the instructions of a given program, many hardware (processor) and software (compiler) techniques have been developed.

In the sequel, we first underline the main algorithmic difference between CompHorner and DDHorner: CompHorner and DDHorner perform essentially the same floating point operations, but the renormalization steps required for double-double computations are avoided in CompHorner. Next, we will prove that CompHorner presents as a consequence more ILP, which certainly explains its better practical performance on modern superscalar processors. We perform our analysis assuming that no FMA is available, that is considering only the implementations of CompHorner and DDHorner described in Section III.

A. Comparison between CompHorner and DDHorner

Let us now compare algorithms CompHorner and DDHorner. For this purpose, we consider below a slightly modified version of the compensated Horner algorithm. Compared to Algorithm 6, we only reorder the floating point operations involved in the computation of \( c_i \) with respect to \( c_{i+1} \), \( x \), \( \pi_i \) and \( \sigma_i \),

```plaintext
function r = CompHorner’(P, x)
    \( r_n = a_i; c_n = 0 \)
    for \( i = n-1 : -1 : 0 \)
        \( [a_i, \pi_i] = \text{TwoProd}(r_{i+1}, x) \)
        \( t = c_{i+1} \oplus x \oplus \pi_i \)
        \( [r_i, \sigma_i] = \text{TwoSum}(p_i, a_i) \)
        \( c_i = t \oplus \sigma_i \)
    end
    \( r = r_0 \oplus c_0 \)
```

Comparing the previous algorithm with Algorithm 7, it is clear that CompHorner and DDHorner perform all the same floating point operations but the renormalization steps needed in algorithm DDHorner. The lines \( [ph_i, pl_i] = \text{FastTwoSum}(th_i, tl_i) \) and \( [sh_i, sl_i] = \text{FastTwoSum}(th_i, tl_i) \) in DDHorner are avoided in CompHorner and so this save 6n flops. Saving 6n flop is clearly not sufficient to explain the practical performance of CompHorner compared to DDHorner. Nevertheless, we will see in the next subsections that thanks to the suppression of the renormalization steps, the compensated evaluation algorithm introduces more ILP than DDHorner.

B. IPC and the ideal processor

A way to evaluate the ILP available in a given program is to compute its IPC (instruction per clock) on an ideal processor [10, p.240]. The IPC of a program, running on a given processor, is the average number of instructions of this program executed per clock cycle. An ideal processor is one where all artificial constraints on ILP are removed. The only limits in such a processor are those imposed by the actual data flows through either register or memory. More precisely, we assume that:

- the processor can execute an unlimited number of independent instructions in the same clock cycle;
all but true data dependencies are removed: any instruction in the program execution can be scheduled on the cycle immediately following the execution of the predecessor one on which it depends;
- branches are perfectly predicted: all conditional branches are predicted exactly;
- memory accesses are also perfect: all loads and stores always complete in one clock cycle.

These assumptions mean that this ideal processor can execute arbitrarily many operations in parallel, and that any sequence of dependent instructions can execute on successive cycles. In the sequel, we will refer to the IPC of a program running on such an ideal processor as its ideal IPC.

Now, let us study algorithm CompHorner running on this ideal processor when evaluating a polynomial of degree $n$:

C. Ideal IPC of CompHorner

We consider on Fig. 2.a the data-flow graph iteration $i$ of the main loop of algorithm CompHorner. This data-flow graph is based on the C code implementation of CompHorner provided in Section III. The inputs in square boxes are critical inputs since they are the outputs of the previous iteration $i + 1$. We can distinguish three critical paths of interest (represented with dashed edges) in this data-flow graph:

- one from $r_{i+1}$ to $r_i$ containing 2 instructions,
- one from $r_{i+1}$ to $c_i$ containing 10 instructions,
- one from $c_{i+1}$ to $c_i$ containing 2 instructions.

Since the critical path from $r_{i+1}$ to $c_i$ has length 10, the whole iteration can be executed within 10 cycles with the ideal processor. From these remarks, we represent on Fig. 2.b the execution of iteration $i$ as a box of length 10 cycles, where:

- $r_{i+1}$ is consumed at the first cycle of the iteration,
- $r_i$ is produced at cycle 2,
- $c_{i+1}$ is consumed at cycle 8,
- and $c_i$ is produced at cycle 10.

On Fig. 2.c, we represent the execution of the $n$ iterations on the ideal processor. We can see that one iteration starts every two cycles, and that two successive iteration executions overlap by 8 cycles. We deduce that the latency of the whole loop of algorithm CompHorner is $2n + 8$ cycles. Since the whole loop requires $22n$ floating point instructions, the ideal IPC for the loop of algorithm CompHorner is

$$IPC_{CompHorner} = \frac{22n}{2n + 8} \approx 11 \text{ instructions per cycle.}$$

D. Ideal IPC of DDHorner

On Fig. 3, we perform the same analysis to determine the total latency of DDHorner execution. We represent the data-flow graph for iteration $i$ of algorithm DDHorner on Fig. 3.a. From the analysis of this data-flow graph, we represent on Fig. 3.b the execution of iteration $i$:

- $sh_{i+1}$ is consumed at the first cycle of the iteration,
- $sh_i$ is produced at cycle 17,
- $sl_{i+1}$ is consumed at cycle 3,
- and $sl_i$ is produced at cycle 19.
From these remarks, we represent on Fig. 3.c the execution of $n$ iterations on the ideal processor. One iteration starts every 17 cycles, and two successive iteration executions only overlap by 2 cycles. Therefore, the latency of the whole loop of algorithm DDHorner is $17n + 2$ clock cycles on the ideal processor. Since the loop execution requires $28n$ floating point instructions, the IPC of DDHorner running on the ideal processor is

$$IPC_{DDHorner} = \frac{28n}{17n + 2} \approx 1.65 \text{ instructions per cycle.}$$

### E. Analysis

The ideal IPC of CompHorner is therefore much greater than the one of DDHorner:

$$IPC_{CompHorner} \approx 6.66 \times IPC_{DDHorner}.$$  

Clearly this means that more ILP is available in CompHorner than in DDHorner. We stress that this theoretical analysis cannot explain in a quantitative manner the actual ratios reported in Table III. Indeed, measured ratios are from real processors with limited resources while ideal IPC is computed.
assuming a processor which exploits all the ILP available in the algorithms. However this certainly explains in a qualitative manner the better efficiency of the compensated algorithm on modern processors designed for exploiting ILP.

The consequence of the renormalization steps needed for double-double computations also appears clearly if we compare Fig. 3 to Fig. 2. They act has “bottlenecks” during the execution of DDHorner. The three floating point operations involved in every renormalization steps are represented in boxes on Fig. 3. If we compare one iteration of the loop of DDHorner to one iteration of CompHorner, it appears that:

- the latency of every iteration of DDHorner is larger because of the first renormalization step —cycles 10 to 12 on Fig. 3(a),
- due to the second renormalization step —cycles 17 to 19 on Fig. 3(a)— the overlap between two consecutive iterations of DDHorner is smaller.

The fact that CompHorner avoids any renormalization step is therefore the reason why it exhibits more ILP.

V. CONCLUSION

Compensated Horner algorithm yields more accurate polynomial evaluation than the classic Horner iteration. Its accuracy is similar to a Horner iteration performed in a doubled working precision. Compensated Horner evaluation also is very efficient, since it runs at least twice as fast as the double-double Horner counterpart still providing the same output accuracy.

We summarize our analysis as follows. Avoiding the renormalization steps needed for double-double computations, the compensated Horner algorithm presents more ILP than its counterpart using double-double arithmetic. In our point of view, this gives a qualitative explanation of its practical efficiency.

The same conclusion certainly holds for other compensated algorithms, that also avoids the renormalization steps. This is the case for:

- the improvements of the compensated Horner algorithm when a FMA is available [9],
- compensated triangular system solver presented in [18],
- compensated summation and dot product in [7].

Let us also emphasis that ILP analysis, as described in this paper, may also be very useful to explain and compare the efficiency of many other numerical algorithms.

The error-free transformations TwoSum and TwoProd are the keys to improve the precision of floating point computation, either with double-double arithmetic or compensated algorithms. Solutions to facilitate their portable implementation have been discussed during the current IEEE-754 revision work [19]. Let us cite the “tail operations” or the ADD3 operator —for $a, b, c \in \mathbb{F}$, ADD3 = fl$(a + b + c)$. Unfortunately these new operators are not anymore in the draft. Nevertheless the current revision draft proposes the standardization of the FMA and of operations minNumMag and maxNumMag —for $a, b \in \mathbb{F}$, if $|a| \leq |b|$ then minNumMag$(a,b) = a$, or $b$ otherwise. These additional features will be useful to implement more efficiently the EFT for the multiplication and the addition within the TwoProd with FMA and the FastTwoSum algorithms.

ACKNOWLEDGMENT

The authors thank B. Goossens and D. Parello for stimulating discussions about performance analysis.
### Environment (III)

<table>
<thead>
<tr>
<th>deg</th>
<th>H</th>
<th>CH</th>
<th>DDH</th>
<th>time in clock cycles</th>
<th>Normalized time in execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>255</td>
<td>578</td>
<td>1373</td>
<td>1.0</td>
<td>2.3</td>
</tr>
<tr>
<td>15</td>
<td>323</td>
<td>780</td>
<td>1980</td>
<td>1.0</td>
<td>2.4</td>
</tr>
<tr>
<td>20</td>
<td>383</td>
<td>957</td>
<td>2588</td>
<td>1.0</td>
<td>2.5</td>
</tr>
<tr>
<td>25</td>
<td>428</td>
<td>1170</td>
<td>3203</td>
<td>1.0</td>
<td>2.7</td>
</tr>
<tr>
<td>30</td>
<td>501</td>
<td>1373</td>
<td>3810</td>
<td>1.0</td>
<td>2.7</td>
</tr>
<tr>
<td>35</td>
<td>563</td>
<td>1568</td>
<td>4418</td>
<td>1.0</td>
<td>2.8</td>
</tr>
<tr>
<td>40</td>
<td>622</td>
<td>1808</td>
<td>5033</td>
<td>1.0</td>
<td>2.9</td>
</tr>
<tr>
<td>45</td>
<td>683</td>
<td>1988</td>
<td>5679</td>
<td>1.0</td>
<td>2.9</td>
</tr>
<tr>
<td>50</td>
<td>735</td>
<td>2183</td>
<td>6285</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>55</td>
<td>803</td>
<td>2378</td>
<td>6893</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>60</td>
<td>863</td>
<td>2588</td>
<td>7500</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>65</td>
<td>908</td>
<td>2776</td>
<td>8115</td>
<td>1.0</td>
<td>3.1</td>
</tr>
<tr>
<td>70</td>
<td>983</td>
<td>2978</td>
<td>8723</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>75</td>
<td>1034</td>
<td>3173</td>
<td>9338</td>
<td>1.0</td>
<td>3.1</td>
</tr>
<tr>
<td>80</td>
<td>1103</td>
<td>3375</td>
<td>9945</td>
<td>1.0</td>
<td>3.1</td>
</tr>
<tr>
<td>85</td>
<td>1163</td>
<td>3570</td>
<td>10560</td>
<td>1.0</td>
<td>3.1</td>
</tr>
<tr>
<td>90</td>
<td>1215</td>
<td>3765</td>
<td>11167</td>
<td>1.0</td>
<td>3.1</td>
</tr>
<tr>
<td>95</td>
<td>1283</td>
<td>3960</td>
<td>11779</td>
<td>1.0</td>
<td>3.1</td>
</tr>
<tr>
<td>100</td>
<td>1343</td>
<td>4163</td>
<td>12383</td>
<td>1.0</td>
<td>3.1</td>
</tr>
<tr>
<td>105</td>
<td>1388</td>
<td>4358</td>
<td>12998</td>
<td>1.0</td>
<td>3.1</td>
</tr>
<tr>
<td>110</td>
<td>1465</td>
<td>4560</td>
<td>13605</td>
<td>1.0</td>
<td>3.1</td>
</tr>
<tr>
<td>115</td>
<td>1523</td>
<td>4768</td>
<td>14220</td>
<td>1.0</td>
<td>3.1</td>
</tr>
<tr>
<td>120</td>
<td>1583</td>
<td>4960</td>
<td>14828</td>
<td>1.0</td>
<td>3.1</td>
</tr>
<tr>
<td>125</td>
<td>1643</td>
<td>5153</td>
<td>15435</td>
<td>1.0</td>
<td>3.1</td>
</tr>
<tr>
<td>130</td>
<td>1695</td>
<td>5348</td>
<td>16055</td>
<td>1.0</td>
<td>3.2</td>
</tr>
<tr>
<td>135</td>
<td>1763</td>
<td>5543</td>
<td>16655</td>
<td>1.0</td>
<td>3.1</td>
</tr>
<tr>
<td>140</td>
<td>1823</td>
<td>5745</td>
<td>17265</td>
<td>1.0</td>
<td>3.2</td>
</tr>
<tr>
<td>145</td>
<td>1890</td>
<td>5939</td>
<td>17880</td>
<td>1.0</td>
<td>3.1</td>
</tr>
<tr>
<td>150</td>
<td>1943</td>
<td>6134</td>
<td>18488</td>
<td>1.0</td>
<td>3.2</td>
</tr>
<tr>
<td>155</td>
<td>2003</td>
<td>6329</td>
<td>19095</td>
<td>1.0</td>
<td>3.2</td>
</tr>
<tr>
<td>160</td>
<td>2079</td>
<td>6523</td>
<td>19710</td>
<td>1.0</td>
<td>3.1</td>
</tr>
<tr>
<td>165</td>
<td>2138</td>
<td>6728</td>
<td>20318</td>
<td>1.0</td>
<td>3.1</td>
</tr>
<tr>
<td>170</td>
<td>2190</td>
<td>6923</td>
<td>20925</td>
<td>1.0</td>
<td>3.2</td>
</tr>
<tr>
<td>175</td>
<td>2252</td>
<td>7118</td>
<td>21540</td>
<td>1.0</td>
<td>3.2</td>
</tr>
<tr>
<td>180</td>
<td>2318</td>
<td>7317</td>
<td>22149</td>
<td>1.0</td>
<td>3.2</td>
</tr>
<tr>
<td>185</td>
<td>2372</td>
<td>7515</td>
<td>22775</td>
<td>1.0</td>
<td>3.2</td>
</tr>
<tr>
<td>190</td>
<td>2439</td>
<td>7718</td>
<td>23367</td>
<td>1.0</td>
<td>3.2</td>
</tr>
<tr>
<td>195</td>
<td>2499</td>
<td>7913</td>
<td>23968</td>
<td>1.0</td>
<td>3.2</td>
</tr>
<tr>
<td>200</td>
<td>2565</td>
<td>8115</td>
<td>24585</td>
<td>1.0</td>
<td>3.2</td>
</tr>
</tbody>
</table>

**Average overhead:** 2.8% | **Average overhead:** 2.8%


### Environment (VII)

| 10 | 386 | 448 | 793 | 1.0 | 1.2 | 2.1 |
| 15 | 406 | 488 | 1013 | 1.0 | 1.2 | 2.5 |
| 20 | 434 | 528 | 1233 | 1.0 | 1.2 | 2.8 |
| 25 | 457 | 568 | 1453 | 1.0 | 1.2 | 3.2 |
| 30 | 474 | 608 | 1673 | 1.0 | 1.3 | 3.5 |
| 35 | 506 | 648 | 1893 | 1.0 | 1.3 | 3.7 |
| 40 | 514 | 688 | 2113 | 1.0 | 1.3 | 4.1 |
| 45 | 537 | 728 | 2333 | 1.0 | 1.4 | 4.3 |
| 50 | 554 | 768 | 2553 | 1.0 | 1.4 | 4.6 |
| 55 | 577 | 808 | 2773 | 1.0 | 1.4 | 4.8 |
| 60 | 594 | 848 | 2993 | 1.0 | 1.4 | 5.0 |
| 65 | 617 | 888 | 3213 | 1.0 | 1.4 | 5.2 |
| 70 | 634 | 928 | 3433 | 1.0 | 1.5 | 5.4 |
| 75 | 666 | 968 | 3653 | 1.0 | 1.5 | 5.5 |
| 80 | 674 | 998 | 3873 | 1.0 | 1.5 | 5.7 |
| 85 | 697 | 1048 | 4093 | 1.0 | 1.5 | 5.9 |
| 90 | 714 | 1089 | 4313 | 1.0 | 1.5 | 6.0 |
| 95 | 737 | 1128 | 4533 | 1.0 | 1.5 | 6.2 |
| 100 | 754 | 1168 | 4753 | 1.0 | 1.5 | 6.3 |
| 105 | 772 | 1208 | 4973 | 1.0 | 1.6 | 6.4 |
| 110 | 794 | 1248 | 5193 | 1.0 | 1.6 | 6.5 |
| 115 | 826 | 1288 | 5413 | 1.0 | 1.6 | 6.6 |
| 120 | 834 | 1328 | 5633 | 1.0 | 1.6 | 6.8 |
| 125 | 857 | 1368 | 5853 | 1.0 | 1.6 | 6.8 |
| 130 | 874 | 1408 | 6073 | 1.0 | 1.6 | 6.9 |
| 135 | 897 | 1448 | 6293 | 1.0 | 1.6 | 7.0 |
| 140 | 914 | 1488 | 6513 | 1.0 | 1.6 | 7.1 |
| 145 | 937 | 1528 | 6733 | 1.0 | 1.6 | 7.2 |
| 150 | 954 | 1568 | 6953 | 1.0 | 1.6 | 7.3 |
| 155 | 986 | 1609 | 7173 | 1.0 | 1.6 | 7.3 |
| 160 | 1004 | 1648 | 7393 | 1.0 | 1.7 | 7.4 |
| 165 | 1017 | 1688 | 7613 | 1.0 | 1.7 | 7.5 |
| 170 | 1034 | 1728 | 7833 | 1.0 | 1.7 | 7.6 |
| 175 | 1057 | 1768 | 8053 | 1.0 | 1.7 | 7.6 |
| 180 | 1074 | 1808 | 8273 | 1.0 | 1.7 | 7.7 |
| 185 | 1097 | 1848 | 8493 | 1.0 | 1.7 | 7.7 |
| 190 | 1114 | 1888 | 8713 | 1.0 | 1.7 | 7.8 |
| 195 | 1146 | 1928 | 8933 | 1.0 | 1.7 | 7.8 |
| 200 | 1154 | 1968 | 9153 | 1.0 | 1.7 | 7.9 |

Average overhead: 2.8 3.5

### References


